How to Integrate Domain-Specific Languages into the Game Development Process

Robert Walter
University of Duisburg-Essen
Forsthausweg 2
Duisburg, Germany
robert.walter@uni-due.de

Maic Masuch
University of Duisburg-Essen
Forsthausweg 2
Duisburg, Germany
maic.masuch@uni-due.de

ABSTRACT
Domain-specific languages make the relevant details of a domain explicit while omitting the distracting ones. This implies many benefits regarding development speed and quality as well as the exchange of information between expert groups. In order to utilize these benefits for game development, we present a language engineering workflow that describes best practices to identify a reasonable domain abstraction, illustrated by means of a language for 2D Point &Click Adventures. We discuss how this process can be integrated into an agile, iterative development process and what thereby needs to be considered.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.11 [Software Engineering]: Software Architectures—domain-specific architectures, languages; D.2.13 [Software Engineering]: Reusable Software—domain engineering

General Terms
Languages

Keywords
Domain-specific language, model-driven software development, language engineering, game development

1. INTRODUCTION
The development of digital games is a multi-tier, creative process in which experts from different disciplines (Art, Design, Writing, Programming, Acting, Music, etc.) need to join forces. Hence, game development combines the artistic challenges of multimedia productions with the engineering challenges of IT-productions [11]. Thereby, developer studios strive to create something unique (be it a new art style, a new gameplay experience or feature, a new way of storytelling, or a combination of these elements) to distinguish their game from others. In order to succeed and to control this process, more and more game studios apply agile development methodologies like Scrum and work highly iterative [5, 10, 15]. A crucial phase within this development cycle is the translation of defined game design elements into the actual game implementation. The quality of the game experience depends on how well the art style, gameplay, and story can be captured within the game.

In this paper, we introduce domain-specific languages (DSLs) as a tool to support the transition of game design to implementation. As a foundation, we briefly introduce DSLs in general. We discuss related research approaches that introduce generic DSLs and the usage of language workbenches for game development to contrast them with our approach of a language engineering workflow for game development. We present the workflow tier by tier and illustrate each step by means of an example language for 2D Point&Click Adventures. In doing so, we show how the language engineering process can be embedded in the game design process, fostering communication between designers and developers inside the problem space. In the concluding section, we discuss how DSLs can be integrated into an agile game development process, fostering iterative processes like Rapid Prototyping and Extreme Programming. Thereby, we likewise elaborate the occurring chances and challenges.

We close this paper with an outline of upcoming tasks.

2. DOMAIN SPECIFIC LANGUAGES
DSLs have a long history in software engineering and have been implemented for various use cases [22]. In recent years, they have become more established, due to the success of model-driven software development (MDSD). In order to have a common understanding of what a DSL is, we start with the definition by Fowler [8]:

[A DSL is] a computer programming language of limited expressiveness focused on a particular domain.

The purpose of a DSL is to let users write computer instructions in a more natural way. Compared with general purpose languages (GPLs) like C or Java, DSLs provide a limited vocabulary and almost no control structures. In short, a DSL features exactly the elements necessary to describe its domain, which makes it easier to learn and less expressive than a GPL. Like GPLs, DSLs provide a formal
syntax. Hence, a domain-specific program (DSP) can be processed automatically, for example with an interpreter or generator.

DSLs are commonly subdivided into internal and external DSLs [8]. An internal DSL is embedded in a GPL, merely using a limited set of the features of the GPL. Hence, internal DSLs do not have an individual grammar and thus do not need a distinct tool set to be processed. As a consequence, an internal DSL is bound to the grammar of its host language, which constraints the flexibility in syntax design. A good example for an internal DSL is incorporated in the Google Guice framework\(^2\). It provides a language-like Java API to utilize the dependency injection pattern [6]. The most common way to define an internal DSL is through combined methods (functions, or procedures, depending on the host language). Listing 1 illustrates the Method Chaining Pattern [8] to create an internal DSL. Using Guice, a “service” interface is bound to a “service” implementation as a singleton. Note that this is plain Java code.

```java
1 bind(Service.class)
2 .to(ServiceImpl.class)
3 .in(Scope.SINGLETON);
```

**Listing 1: Internal DSL example (Method Chaining Pattern)**

Internal DSLs exploit the advantages of language-like programming APIs. Hence, internal DSLs address exclusively target groups with an at least basic programming background. In the game development context, we consider internal DSLs as a useful technique to simplify development processes, since many domain experts (Game Designer, Sound Engineer, etc.) have a technical background and thus—even if they are not intended to use internal DSLs to develop game logic—are able to understand corresponding programs. We examine this in Section 5 in more detail.

External DSLs are self-contained languages. This means, that they feature a distinct grammar. Due to that, an external DSL can be tailored exactly to the requirements of the domain, using semantically meaningful keywords and grammar rules. Similar to GPLs, a program written in an external DSL is processed in the form of an abstract syntax tree (AST). As a consequence, one needs to define proper processing tools in order to use an external DSL. The creation of such tools is increasingly supported by language workbenches [7] like Xtext [3] or MPS\(^3\). Popular examples for external DSLs are regular expressions and SQL.

Aside from their environment, DSLs differ in their technical orientation. The internal DSL integrated in Guice, for example, is a technical-oriented DSL. It utilizes the dependency injection pattern, which can be used for arbitrary application development. SQL is an example for a technical-oriented external DSL. Altogether, we categorize DSLs that abstract a technical problem space as technical-oriented DSLs (compare [12]). DSLs of this category trivially address target groups with programming skills. In contrast, application- and business-oriented DSLs comprise languages that describe non-technical domains. For example, Inform\(^4\), currently in its seventh installment, features a declarative DSL with a very natural language to create interactive fiction applications. Inform 7 first and foremost addresses writers, newcomers as well as professionals, who need no programming experience to create an interactive fiction application.

Listing 2: Rule description in Inform 7

Some DSLs might blur the boundaries between technical and non-technical DSLs. However, we presented this differentiation to underline the linear relation between the technical orientation of a DSL and the technical experience of its target group. We will refer to this in sections 4 and 5.

Before we discuss available approaches of DSLs in game development in the next section, we would like to draft an example scenario to make their application value more vivid: Think of a DSL for branching dialogues, which is tailored to a game writer’s needs with the corresponding tooling tailored to a game studio’s technical and procedural requirements. Thus, the game writer (the domain expert) can develop the dialogues within a natural environment. All subsequently depending artifacts, like actor scripts sorted by character for audio recording, the necessary data format for the story engine, or a dialogue tree representation, can then be derived at least semi-automatically with the aid of proper generators. The artifacts written by the game writer become the common source and documentation for the dialogues of a game, which makes the propagation of changes very simple. It is also possible to integrate semantic constraints into the language, so that dialogues fulfill individual system requirements by definition (e.g. define that the name of a speaking character is always stated in capitals in the final game). The manual integration processes of dialogues into a game, which are often nothing but time-consuming, error-prone copy/paste tasks, would no longer be necessary.

### 3. DSLs in Game Development

Scripting languages like LUA or Python play an important role in game development. Despite their comparably small syntax, they are not DSLs, since they are not created to define a distinct problem space and therefore are not further examined in this paper.

Besides the already mentioned language in Inform 7, the Text Adventure Development System\(^5\) (TADS) features another language for the definition of interactive fiction games. The MOO language\(^6\) and LPC\(^7\) are similar languages to create multi-user dungeons (MUDs). What these languages have in common is that they are platform-specific languages for a distinct genre, i.e. they are dedicated to their corresponding target platform and characteristic gameplay. The <e-Game> project [14] also fits into this category. The developers pursued a documental approach which uses an XML-based markup language to enrich storyboards for educational adventure games. This means that game writers can

---

1. We introduce the term domain-specific program to refer to a program written in a DSL.
2. [http://code.google.com/p/google-guice/](http://code.google.com/p/google-guice/)
enhance a storyboard with semantically meaningful markers which are then processed by the corresponding engine. At first glance, this might sound closely related to our presented work, especially since we use a DSL that describes a similar genre as an illustrating example. However, there are several fundamental differences that we would like to point out. First, the Game is an instance that shows how an external, non-technical DSL can be applied in a distinct game development context. We may use a similar language, but only for the purpose of illustrating our actual work, namely a language engineering workflow that describes how development studios can incorporate an arbitrary DSL into their existing development process. In addition, as the transition from the Game project to the Adventure authoring environment [21] underlines, the documental approach targets to enable one specific, non-technical expert group (game writers) to create small but runnable games within a distinct genre. This is especially interesting in a context like prototyping or authoring for serious games (e.g. e-learning games). We, in contrast, examine the advantages and disadvantages of DSLs in a broader scope (covering also the application of technical-oriented and internal DSLs) with the focus of integrating them into existing professional development processes.

Another commonly pursued approach is to create a DSL that enables the formal description of games on a very high level of abstraction, with the intention to make it applicable for a broad range of games. In [18], Reyno and Cubel introduce an approach to specify gameplay with a platform-independent model. They ground their work on game definitions by Salen and Zimmerman [20], Djaouti et al. [4], and others. Their central objective is to replace gameplay descriptions written in natural language with a platform-independent model in order to be able “to define and communicate gameplay at a high level of technical abstraction”. Their approach might be reasonable in the very early stages of gameplay modeling. However, we consider it not practicable. In order to cover as many of the individual requirements of every game studio respectively of every game as possible, these languages become very generic and thus less precise. It is also unclear how the models are meant to be processed. In contrast, we pursue the goal to let developer studios create individual languages tailored to their particular demands in an efficient, effective and flexible manner.

Another approach by Reyno and Cubel is presented in [19]. The authors introduce both platform independent as well as platform specific meta-models to formally describe 2D platform game prototypes. The provided UML models offer an abstract syntax for the given context. However, the authors do not provide concepts for a concrete syntax, which is necessary to utilize the models for a target group.

An approach that combines a software factory with visual DSLs and generators to create adventure games is presented by Furtado and Santos [9]. They provide a “Game Modeling DSL” as well as a “HUD Creation DSL” to specify an adventure game. They developed the DSLs according to a methodology presented in [22]: analysis, implementation (which they enhance by two additional tasks), and use.

In [13], Maier and Volk discuss language workbenches in order to facilitate language-oriented game development. They argue that language workbenches reduce the set-up costs for software factories as well as the development time of games.

In contrast to [18, 19, 9], our approach does not introduce a generic language for game development. In contrast, we understand the term domain-specific in a more narrowed down scope. We think that the agile nature of game development in combination with the individuality of every developer studio requires languages that fit into their very specific environment. Hence, our language engineering workflow—along with language workbenches or other tools of choice—aims for a feasible and flexible creation, optimization, and application of DSLs, integrated into an iterative development process. It could be argued that this approach contradicts with the idea of language reusability and would therefore lead to a tremendous amount of languages, making it inapplicable. However, our workflow does not imply that created DSLs might not be reused in subsequent projects. But, as we describe in sections 4 and 5 in more detail, we consider reusability to be a natural and organic process that evolves with the growing experience of developers working with DSLs and therefore does not need to be fostered artificially.

While we conform to the findings in [13] regarding the benefits of language workbenches, our approach is knowingly independent from the way a DSL tool set is built, which means that the tools (e.g. parser, generator, editor) can be implemented manually, with the help of a parser generator [17], as well as supported by a language workbench.

4. LANGUAGE ENGINEERING WORKFLOW

In order to facilitate the process of designing a DSL and afterwards creating the corresponding tool set, we established a flexible yet stable bottom-up workflow (Figure 1). It can be used to define any kind of DSL (internal and external) with any degree of technical orientation.

![Figure 1: DSL Engineering Workflow](image-url)

It is important to understand the positioning of the workflow in context of the game development process. The workflow comprises three plus one tiers. The three lower tiers represent the actual language engineering process. This process becomes part of the game design phase. This means that, in an iterative process where the game design is refined stepwise, the used DSLs might also be refined to fit the new
requirements best. As we outline in Section 4.3, DSLs can even be used as a tool for refining game design. The top level tier of the workflow illustrates the interface to the implementation or prototyping phase.

In the following subsection, we introduce two roles that have to participate in our language engineering process. Subsequently, we describe the workflow tier by tier. To illustrate every tier, we created an external DSL which allows game designers to describe and prototype a 2D Point&Click Adventure in a very natural (yet formal) way. Note that this language only serves as workflow illustration. It does not define how 2D Point&Click Adventures should be defined in general. For the language tool set, we used the Xtext language framework in combination with the Modeling Workflow Engine (MWE, [1]) and the Xpand template language [2]. The DSPs created in the language are finally translated to a runnable XNA game.

4.1 Roles

In [12], the author distinguishes between language user ("the person who simply uses a software language to create an application") and language engineer ("the person who creates software languages"). For our game-oriented workflow, we consider these two roles to be more tightly intertwined and therefore propose another assignment of roles.

![Role Assignment of DSL Engineering Workflow](image)

Figure 2: Role Assignment of DSL Engineering Workflow

First, we define domain expert as a person with extensive domain knowledge and expertise. The domain expert is in charge of identifying and defining the entities and semantics of the domain. Depending on the target platform and further processing of the DSPs, the domain expert might also participate in designing the back-end of the tool set. However, he or she is not in charge of developing any tools for language processing. For this responsibility, we redefine language engineer as a person with extensive knowledge in language design and tool creation for various environments and purposes. Common tools that have to be built for working with a DSL are editors, transformers, executors and analyzers (compare [12]). Depending on the scope of the domain, it is reasonable that more than one domain expert or language engineer participate in the language engineering process. Furthermore, the described tasks of a domain expert could be sub-divided in front-end and back-end tasks. This means that there is a front-end domain expert, for example a game writer, who defines the features of a story-writing language. A first back-end domain expert, a developer of the story engine, covers the design of the intended outcome and further processing, while another back-end domain expert, a professional voice actor, determines which information is necessary for their work. The language engineers take this information and synthesize it in a language definition and a corresponding tool set.

4.2 DSL Requirements

When defining a new language, the workflow starts with the definition of the language requirements. The requirements are determined by domain characteristics, which means that we need to find a proper abstraction of the domain. The domain characteristics should at least comprise the central domain entities and central terms with clearly defined semantics for the given domain. As described in Section 4.1, this process should be led by domain experts. Language engineers should interpret their role in a predictive, hence moderating way during this phase in order to create both a sound domain outline and a solid foundation for the forthcoming tasks. In addition, general criteria, like general language development guidelines we used in our development (see Table 1), also influence the DSL requirements. As the bidirectional relationship between the DSL requirements and the reference model points out, we propose a small iterative process between these two artifacts. Thus, the DSL requirements are meant to capture the domain so that both domain experts and language engineers are aware of the vocabulary, purpose, scope, and target group of the domain.

For the Adventure DSL example, we initially identified basic entities to describe the visible structure of 2D Point&Click Adventures.

- **Room** describes a self-contained area which can be entered and left by the player character through doors; it is illustrated by a background image and contains (optional) background music as well as an arbitrary amount of objects and characters.
- **Door** describes a portal to change the room in which the player character is; a door has a position, always belongs to two rooms, is either in state "opened" or "closed", and thus has at least two sprites to illustrate the corresponding state.
- **Character** describes the playable and non-playable characters in the game, illustrated by at least one sprite; a character is always assigned to exactly one room and one position at a time.
- **Object** describes useable and non-useable objects in a room; an object is also illustrated by at least one sprite, assigned to a distinct room and a position.
- **Inventory** describes a virtual place in which the player manages all collected objects.
- **Action-Verb Menu** describes a set of interaction possibilities the player can apply to objects and characters.
- **Context Information** provides information about the position the player points to.
Note that this first visual consideration abstracts from technical issues. We deliberately omitted aspects like file formats and game resolution in order to shape an independent domain description. With that in mind, we also defined gameplay-related entities.

- **Task** describes puzzles and challenges the player faces during the game.
- **Goal** describes a state the player has to reach in order to gain further progress.
- **Constraint** checks if goals are achieved in order to describe causal relationships between tasks.
- **Action** describes an action that has to be performed by the game; it is either triggered by the player or by the game.
- **Dialogue** describes verbal interaction between the player character and non-player characters.

Again, we omitted issues like character movement or dialogue implementation since they are of no concern for game designers.

As mentioned, we enhanced our DSL requirements with language engineering guidelines taken from [12]. In addition, we considered two criteria of the software-ergonomic principles of system design, stated in EN ISO 9241-110, namely self-descriptiveness and error tolerance.

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplicity</td>
<td>An easy-to-learn language is easy to use</td>
</tr>
<tr>
<td>Consistency</td>
<td>Choose a suitable design and hold [on] to it</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Give the user flexibility in how to formulate a DSP</td>
</tr>
<tr>
<td>Homogeneity</td>
<td>Show related concepts in the same way, and keep them together; show unrelated concepts differently and [separately]</td>
</tr>
<tr>
<td>Color</td>
<td>Use color sparsely</td>
</tr>
<tr>
<td>Testing</td>
<td>Test the language design on language users</td>
</tr>
</tbody>
</table>

Table 1: Language Engineering Guidelines

Based on these DSL requirements, we developed a reference model and a reference implementation as described in the following subsection.

### 4.3 Reference Artifacts

We define two ephemeral reference artifacts that have to be created in the second engineering tier. The intention is to find an exemplary instance of the front-end (→ reference model) and the back-end (→ reference output) of the tool set. It might well be reasonable to create more than one reference artifact each in order to cover all DSL requirements.

#### 4.3.1 Reference Model

The reference model is an example program written in the intended language. The central issue is to think of a representation that covers all DSL requirements in a target group-oriented way, which makes this a very front-end-oriented process. Thereby, the reference model serves as communication platform between domain experts and language engineers in order to combine and manifest their expertise in the reference model and thus in the final tool set.

In other words, the reference model is the vehicle to map the DSL requirements to the concrete syntax of the language. Again, the language engineers should moderate the creation of the reference model, always keeping the back-end in mind.

We also used the reference model as a brainstorming tool and propose to incorporate this into the game design process when working with DSLs. The task of finding a representation that separates a game concept from its technical realization in a multi-expert group implicitly leads to a reconsideration of existing design ideas and might even create new approaches.

Listing 3 shows a reference model excerpt for our Adventure DSL. First, a room named cafeteria is created. Its background image is defined by a relative path to an image file and its background music by a path to a sound file.

```
1 The cafeteria is a room.
2 Its background image is "Sprites/cafeteria.png".
3 The background music of the cafeteria is "Sounds/cafeteria.wav".
4
5 The fork is an object.
6 Its image is "Sprites/fork.png".
7 Its position is (400,300). It cannot be picked up.
8 Its display name is "A dirty fork". It is in the cafeteria.
```

Listing 3: Reference Model Excerpt (a)

Afterwards, a fork is defined as an object. It is placed in the cafeteria at a distinct position with the initial property that it cannot be picked up by the player.

In order to provide a simple language, we defined short, clear sentences (Simplicity). The user should be able to refer to the last created element using possessive ("Its") and personal ("It") pronouns, but it should not be mandatory (Flexibility). As illustrated in lines 2–3, for example, related concepts are expressed in related ways (Homogeneity). An issue that we discuss in Section 5 in more detail is the arrangement of objects and characters using a mere textual language. So far, we use Cartesian coordinates (compare line 7) for positioning. Listing 4 shows another excerpt from the reference model.

First, the character Professor Einstein is created. Note that, compared to the previously defined fork, the room and the position of the character are arranged differently, which shows another instance of flexibility in the language.

```
1 Professor Einstein is a character.
2 His display name is "Prof. Einstein". He is in the cafeteria.
3 His image is "Sprites/einstein.png". His position is (200, 300).
4
5 Dialogue with Professor Einstein:
6 1 "Hello, how are you?" - "I'm hungry!" - Einstein is hungry
7 1.1 "Me too." - "Let's find something to eat."
8 1.2 "Sorry, I have nothing to eat." - "Bye bye."
9 2 "Sorry, I have to go."
10 The dialogue ends here.
```

Listing 4: Reference Model Excerpt (b)

Starting with line 5, a dialogue with Professor Einstein is defined. Each dialogue line starts with a numeric identifier, followed by an option-answer-information tuple. De-
depending on the option the player chooses in the game, the corresponding NPC answer is supposed to be triggered and an information is received, if available. The corresponding lower-level dialogue lines are triggered subsequently. Thus, dialogue trees can be defined. The dialogue ends with the statement in line 10. The structure for writing dialogues that way was developed in an iterative process. Initially, we had no intention to provide branching dialogue since we thought that this would be inconsistent with the idea of a simple language. Creating and refining the reference model stepwise, however, led to the presented structure, which noticeably enriches the gameplay options.

We also utilized the verb-object design paradigm—well-known from several Lucasfilm Games like “Maniac Mansion” or “The Curse of Monkey Island”—as illustrated in Listing 5.

```
Pick up fork (Einstein is hungry):
Say “I won’t take that fork, it’s dirty.”
Pick up fork (enabled):
Say “Thanks, Einstein, you must have been really hungry!”
```

Listing 5: Reference Model Excerpt (e)

In the first line, the combination of pick up and fork is declared as a possible action, only available if the player has gathered the information that Einstein is hungry. Afterwards, it is defined what happens in the game when the player tries to perform the action. So far, the language provides a set of pre-defined action-verbs. Furthermore, the user is able to define own action-verbs whose semantics must also be provided by the user in the generator templates.

After this action in Listing 5, the fork is no longer dirty and the player can finally pick it up as defined in the last reference model excerpt (Listing 6).

```
Forkgoal achieved.
Play “Sound/goalachieved.wav”.
The action ends here.
```

Listing 6: Reference Model Excerpt (d)

4.3.2 Reference Output

In general, DSPs can be processed in two different ways (compare [12], [8]). It is either possible to execute a DSP directly with a dedicated interpreter or to process it with a generator, for example to generate code or configuration files.

In case of generation, the reference output has to represent the desired generated artifacts corresponding to the reference model. In case of interpretation, the reference output has to mirror the desired results and actions described in the reference model. The reference output has to be defined for every target platform and every reference model. That way, the language engineers can derive the processing rules necessary to fill the gap between both abstraction layers. Actually, the gap correlates to the applied degree of abstraction in DSL. In other words, the processing rules have to add the target platform-relevant details that are omitted by the corresponding DSL.

In the Adventure use case, we decided to use Microsoft’s XNA Game Studio as the target platform. Hence, we created a 2D Point&Click adventure game, implemented in C# within the XNA framework that corresponds to our reference model. Although with limited scale, the game contains all features both the DSL as well as the generator are intended to provide.

4.3.3 Related Work

In most cases, there exists related work that can help define the reference artifacts. This can address approaches the target group uses in its daily business. For example, game writers often organize dialogue lines in tuples, using IDs to tag and refer to these tuples. We mapped this structure to our reference model and finally utilized it in the adventure language. Regarding the reusability aspect, related work can also refer to similar DSLs created for a former project, which can expedite the language engineering process very much.

4.4 Language Definition

The language definition is directly derived from the reference model. The language engineers have to identify syntax rules and static semantics based on the reference model. Depending on the technology used to create the language tool set, small adaptations—in comparison to the reference model—are inevitable. However, the language engineers should always confer with the domain experts in order to find an adequate solution. Despite the fact that the language is created in a target group-oriented process, we propose to create rich tool support. Language workbenches support the creation of smart editors, which, for example, feature syntax highlighting, text completion, and syntax analysis by default.

Just like in GPLs, the abstract syntax is the structural description of a DSL. In its representation as an AST, a DSP can be processed by a parser. The user, however, writes DSPs in a concrete syntax, using defined keywords in a distinct notation. The concrete syntax can therefore be seen as an instance of the abstract syntax. This means that it is possible to create several concrete syntax definitions for the same abstract syntax, for example a visual and a textual representation.

We used an EBNF\(^8\)-similar language, provided by Xtext, to define one concrete syntax. Based on this artifact, Xtext creates the abstract syntax as well as a default language editor with syntax highlighting, code completion and other features automatically. Figure 3 shows an excerpt of the reference model written in the final language.

We applied minor changes compared to the reference model. Our language does not support multi-word identifiers like Professor Einstein or Einstein is hungry. Although it would be possible to provide such identifiers, it would negatively affect cross-referencing and code generation. Hence, we decided to only support standard identifiers which might slightly affect the readability of the language.

However, we managed to incorporate the usability as well as structural requirements defined in the reference model in our current language. Thus, our language provides the definition of rooms, characters, doors, objects, action-verbs,...

\(^8\)Extended Backus Naur Form
context information as well as tasks, goals, constraints, actions and branching dialogue, to name the main features.

Figure 3: Example Written in Adventure DSL

4.5 Processing Rules

In case of code generation, processing rules determine how the input model (DSP as AST) is transformed into another form. The output is simply a set of files, for example C# code. In case of direct interpretation, the interpreter rules determine how a DSP command is executed. This means that the type of output can be manifold, ranging from visual responses in the game world to internal data alteration. In both cases, the processing rules have to close the gap between DSL and target platform. To illustrate this, we provide an example in the next paragraph.

An Adventure DSP is transformed to an XNA game by the Xpand generator. A DSP is therefore first translated into its AST form by an ANTLR parser. The parser is created by the Xtext framework automatically. The transformation rules for the Xpand generator have to be provided by generator templates. An Xpand generator template contains three kinds of information: (1) AST-invariant text, which is always exported in the same way; (2) Wildcards, which are replaced during the generation process with information gathered from the AST; (3) Commands and techniques (conditions, loops, polymorphism, functional extensions, aspect oriented programming) to control the generation process depending on the content of the DSP. Listing 7 illustrates some Xpand code.

Listing 7: Xpand Template Excerpt

First, an AST-invariant C#-comment is declared. Then, a loop iterates through all rooms that have been defined in the DSP. For each room, an instance of Room is added to the roomList-object. For example, this template excerpt would lead to the transformation depicted in Listings 8 and 9.

Listing 8: DSP Excerpt

The game designer does not have to care about the internal data structure when defining the rooms in the DSP. That is because it is a detail that has been left out when defining the DSL. Moreover, the example points out how software generation can be used to assure high code quality. Language engineers and back-end domain experts can utilize their software architectural knowledge in generators. This way, the same concepts on DSP-level are always realized the same way. This is also especially beneficial for boilerplate code, which is nothing but error-prone.

5. DISCUSSION AND OUTLOOK

Domain-specific languages have a great potential for game development, but it is important to understand how to unleash this potential best. Instead of searching for a generic language that describes games or a game genre entirely, we interpret the term “domain” in a more narrow way. On the front-end, DSLs need to be tailored to the demands of the dedicated target group, whilst the back-end needs to fit the individual system requirements. To ensure a positive ROI, however, the problem addressed by a DSL must not be too specific. Hence, we introduced a language engineering workflow that facilitates the creation of DSLs as part of the game design process. That way, developers are able to create and optimize languages for their specific needs. The created languages are meant to close the gap between the game design and its implementation. DSLs achieve this through different characteristics. First, by definition, DSLs describe a clearly determined problem space. As a consequence, irrespective of the technical orientation of a DSL and hence the target group, DSPs represent a communication platform where designers and developers are able to create solutions using a common vocabulary. For example, an internal DSL can enable developers to write programs that are easy to understand for designers with no programming skills.
gramming, where designers and developers implement game elements together, is introduced. DSLs can also foster agile development. Through proper code generation, applied changes can be made visible within seconds. Furthermore, the application of DSLs can increase the development speed and quality over time. Regarding speed, languages as well as generators can be reused, which especially applies for game series. Using code generators instead of repeated manual implementation ensures that a concept described in a DSP is always realized the same way, which ensures a consistent level of code quality. DSLs can also be designed for the purpose of Rapid Prototyping. For this use case, it might be feasible to create a more generic language like the presented Adventure DSL or the languages defined in [9].

DSLs also bear some downsides. Like with every new technology, its usage is connected with a training period, that slows down the overall development velocity at the beginning. While every studio should have several domain experts who are able to define their domain characteristics, the role of the language engineer is especially hard to cast. This is especially true for external DSLs, where the corresponding tool set has to be created. However, we think that DSLs can show a positive ROI very fast because of the aforementioned advantages. Although we focused on textual DSLs in this work, a DSL might also use a graphical concrete syntax. Regardless of the representation form, DSLs cannot and should not be used for the entire game development process. This means that the application fields of DSLs in game development are limited.

In our future work, we are looking forward to evaluate the language engineering workflow on larger projects than the presented Adventure DSL in order to improve it. Another object of our current research is the usage of internal DSLs for Extreme Programming and Rapid Prototyping in game development. In order to facilitate the agility of the language engineering process, we also examine projection-based approaches to create modular language sets. That way, we expect a higher degree of language reusability.

6. ACKNOWLEDGMENTS

We would like to thank Katja Neuwald and Robin Meyer for their research and development support and Heiko von Drachenfels for the helpful discussions.

7. REFERENCES


